(4)

Productivity Engineering in the UNIX† Environment

Augmenting Expensive Functions in Macsyma with Lookup Tables

Technical Report

S. L. Graham
Principal Investigator

(415) 642-2059

"The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government."

Contract No. N00039-84-C-0089

August 7, 1984 - August 6, 1987

Arpa Order No. 4871

†UNIX is a trademark of AT&T Bell Laboratories

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION The Regents of the University of California 6b. OFFICE SYMBOL (*If applicable*)	7a. NAME OF MONITORING ORGANIZATION SPAWAR
6c. ADDRESS (*City, State, and ZIP Code*) Berkeley, California 94720	7b. ADDRESS (*City, State, and ZIP Code*) Space and Naval Warfare Systems Command Washington, DC 20363-5100
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA 8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (*City, State, and ZIP Code*) 1400 Wilson Blvd. Arlington, VA 22209	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (*Include Security Classification*)

* AUGMENTING EXPENSIVE FUNCTIONS IN MACSYMA WITH LOOKUP TABLES

12. PERSONAL AUTHOR(S)

* Carl G. Ponder, Richard J. Fateman

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
technical	FROM _____ TO _____	* August 27, 1987	* 18

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

Enclosed in paper.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (*Include Area Code*) / 22c. OFFICE SYMBOL

Augmenting Expensive Functions in Macsyma with Lookup Tables

Carl G. Ponder
Richard J. Fateman
Computer Science Division
University of California
Berkeley, Ca. 94720

August 27, 1987

Abstract

Tabulating the corresponding inputs and outputs to a computer function reduces recomputation to a simple table lookup. This idea has been used by the symbolic algebra systems Maple and SMP, but to a much lesser degree in Macsyma. We report on some experiments which test this idea for certain critical functions in Macsyma. Although the idea holds some promise, some alleged performance improvements may merely represent redistribution of accounting costs. In many cases performance was degraded. We explain why.

1 Introduction

One way to improve program performance is to associate a lookup table with a computer function `f`, to hold pairs `<x,f(x)>` of inputs and outputs to the function. If the same input is given to the function again, the output is found by searching in the lookup table and returning the precomputed result. This requires that `f` always computes the same output for a given input and that `f` has no side-effects.

Maintaining such a table adds an overhead to all computations of `f`. A lookup must be performed prior to actually computing `f`, and if no entry is found one must be made afterward. The table must also occupy some space. The only benefits to performance occur when inputs are repeated. The tradeoff of eliminated computation and table overhead will determine if a net speedup is accomplished.

The tabular approach, sometimes called "memo-ization" is discussed by Bentley [2] and Abelson/Sussman [1], and has been used by the Maple [4] and

SMP [6] systems. Although the methodology has been described and is widely believed to save time, no published evaluations of this feature have appeared. While in isolated examples the benefit is easy to demonstrate, it is also easy to demonstrate cases where it is wasteful of time and space.

In Maple, all system functions callable from the top level are tabulated. In addition, an option *Remember* is provided for user-defined functions. Specifying this option attaches a lookup table to the specified function. Maple 4.0 provides various options for table management, such as whether or not to empty the table upon garbage-collection.

In this paper we will examine the issues of using such a feature in Macsyma and algebraic manipulation systems in general. For sufficiently contrived test cases, the results are positive. For most cases a slight slowdown occurs. In sections 2 and 3 we analyze the requirements of memo functions. In section 4 we describe special issues of interest in Macsyma. In section 5 we describe some experiments.

2 Overview

The two critical parameters affecting the usefulness of the tabulation feature are the frequency of occurrence of repeated inputs to tabulated functions, and the overhead of maintaining the tables.

2.1 Re-Use Frequencies

The first of these parameters is obviously dependent on the nature of the test cases and the ability of the system to map new requests into previously solved cases. Care in the choice of representation and algorithms contribute to this.

The general view of the tabulated version of a function **f** is to replace the form (using Scheme syntax [1]):

```
(define f (lambda(x) <computation>))
```

with

```
(define f (make-tabular (lambda(x)<computation>)))

;; We have omitted hashtable functions:
;; make-table, lookup, and insert!

(define (make-tabular f)
  ;; make-tabular takes one argument:
```

```
;; a function f, of one argument
;; returns a new function which is a tabular form of f
(let ((table (make-table))) ;; set up an empty hashtable
                            ;; this table will be local to f
     (lambda(x)  ;; this is the body of the new function
            (let ((previously-computed-pair (lookup x table)))
                 (cond ((null? previously-computed-pair)
                        ;; call original if needed
                        (let ((result (f x))) ;; compute result
                             (insert! x result table) ;; insert value
in table
                             result)) ;; return result
                       (else
                        (get-value-from-pair
                           previously-computed-pair)))))))
```

As the function is called on different inputs, the lookup table will grow. This tends to slow down subsequent lookups. Well-managed hash tables keep this cost from growing too fast; alternative search structures such as trees do not seem to have any particular advantage here.

Strategies may be used to trim the size of the lookup table, both for storage economy and speed. For example

- The Maple system can partially or totally empty each table whenever a garbage collection of its heap space is performed.
- Some sort of "locality" based scheme might be employed, such as recording the time of last reference or frequency of reference to each entry, and deleting the ones that do not get as much use.
- Discrimination based on the size of the input or the time required to process it might be plausible. That is, we don't enter inexpensive cases in the table at all.

How well these ideas work depends on patterns of system behavior. An ideal oracle might insert into the table only those entries which will be re-utilized, and would delete the entry after last use. Other techniques must suffice: For example, the Maple system will not tabulate "sin(0.01)" but it will tabulate "sin(pi)" for the sine function; a real number parameter suggests that it will be recomputed less often than a symbolic constant

Strategies for deletion of entries will not be explored any further in this paper. We will also not address the long-term storage of tables except to observe that some computations that may be continued over the course of several "runs" of a program. In such cases, preserving the tables between runs – as part of the saved "image" of a program, or explicitly in some persistent data base,

could be worthwhile. Similarly, several processors simultaneously (and perhaps at distributed locations) solving the same suite of problems in a coordinated fashion should probably be able to share valuable tabulated information.

The test cases used will strongly affect the performance of the tabulation scheme. For example, taking successive derivatives of

$$\log(\log(\log(\log(\log(x)))))$$

(the *Logs* benchmark) causes a large number of common subexpressions to be generated and re-differentiated. Taking successive terms of the sequence

$$\phi_0(x) = 1, \quad \phi_h(x) = 1 + \int_0^x \phi^2{}_{h-1}(t)dt \quad h > 0$$

involves far less recomputation.

An early experiment of finding the reduced Grõbner basis [3] of a set of multivariate polynomials has even less apparent redundant computation. One could argue that we have failed to tabulate the appropriate functions for the Grõbner calculation, but in our view it is likely that it is typical of a large class of computation-intensive largely non-redundant calculations that can be specified in Macsyma. Tabulation just won't help. We have dropped this benchmark from our data, but it should be kept in mind that there is a large body of code for which tabulation cannot provide any benefit.

2.2 Lookup Table Efficiency

Clearly an expensive lookup and insertion mechanism would tend to defeat any success of tabulated functions. We use a relatively fast hash-coding scheme, but we must compute hash codes of rather large expressions (mapping "equal" Lisp S-expressions to the same hash number). Since it is feasible to claim that our hashing mechanism could be made faster, the performance projections in section 5.3 provide upper bounds on performance which are *independent* of the form of the lookup table mechanism. We can predict performance with zero-time hashing and lookup.

3 Data Representation and Matching

The number of recomputations detected in a run of a program may be sensitive to the data representations and algorithms used. For example, consider factoring two polynomials identical up to renaming of indeterminates. The results of factoring will likewise be identical up to the renaming. Using literal equivalence

to search in the lookup table will not identify such matches. One possibility is to somehow canonicalize the internal form of expressions so they are immune to change of variable names. The possibilities for finding matches are open-ended: it would be feasible for transformations to help matching be arbitrarily costly. On the other hand, we should not tolerate any lookup more expensive than computing `f` itself.

We will take some pains to observe a relevant upper bound to global improvement by tabulation. If we chose to tabulate a function `f`, we should predict the percentage of time consumed by `f`; if `f` takes only 5% of the time, even if `f` could be computed free we would get only 5% speedup. Thus profiling [9] is important in the global evaluation of this technique.

The nature of the table generated for a given function will depend on the algorithms used as well as the particular test cases. For example, an integration algorithm that calls itself repeatedly on sub-parts of a problem may build up a table rapidly. Whether the stored data will be reused or not may also depend on the algorithm. For example, one which generates new Lisp "`gensym`" variables for each subproblem may never have any duplicates in its table. Some integration programs make extensive use of (perhaps repetitive) differentiation, which builds up a table associated with that function, and so might benefit.

Some algorithms (like cofactor expansion in determinant calculations) may actually be better suited to dynamic-programming solutions. Although subcomputations are still performed, the dynamic programming tableau is accessed in a very specific manner using direct addressing, rather than the haphazard search used by the lookup table. It is significant if one can systematically discard information that is no longer needed, since running out of memory space is normally one contributor to failures of symbolic computations.

The Maple report [4] uses a Fibonacci-number-returning function as an example of using the *remember* option. Of course the storage and computation of Fibonacci numbers is not a serious problem. Although one can dismiss it as a contrived case, it is conceivable that more complex computations might implicitly follow a similar, but more subtle pattern. The Maple report goes on to state that (page 8)

> ... Although the effect is not as spectacular for most functions, it is not unusual for typical programs to be made roughly 30% faster by the judicious use of option *remember*.

Maple has not been tested without hashing [8]. The feeling of the Waterloo group is that it adds a small overhead to computation while producing a large improvement for certain cases. If option *remember* succeeds only because the functions are well-adapted to dynamic programming, then this only reflects the speedup that can be obtained by re-thinking the algorithm and not re-computing already known partial results. It does, however, save in three respects:

- The programmer is saved the linguistic effort needed to reorganize a computation to tabulate appropriate expressions as they are computed.

- The programmer can, without much effort, take advantage of recognized redundancy and "advise" programs – without delving into the interior of their algorithms – to save previously computed results.
- If the dynamic programming tableau would be mostly empty, the hash table would be a more space-efficient form.

4 Issues in Macsyma

The Macsyma system does not currently use any general tabulation feature for system programs. There is a specific table for factorization that is enabled by setting the **savefactors** variable to **true**, which serves as a "memo" feature for the **factor** command. Tabulation can be used for user-defined functions by using *index* or *array* functions. Using the form **f[x] := ...** rather than **f(x) := ...** causes the function **f** to be defined as an array; the elements of the array are computed only on demand and are saved explicitly in the array after they are first computed. Thus only the "necessary" elements of the array are ever entered, and they are only computed once apiece. This corresponds very closely to the properties of the *remember* option. The SMP *projection* construction is similar, but uses a more elaborate pattern match to determine whether a computation or special case applies. [6]

Few Macsyma system programs of any complexity are true functions which compute their values based solely on their input arguments. There are several hundred global flags controlling such issues as whether to attempt numerical evaluation or how to simplify expressions. The settings of variables of all sorts can also affect results. To properly handle flags, the tables must be sensitive to which flag settings are relevant. Three possibilities emerge:

1. Include flags with each table entry and also use them to compute hash codes;
2. Switch to a different table each time a flag is modified or a variable's value is changed.
3. Clear out the tables when a relevant flag is changed.

It is possible to consider writing the program for the function **f** so that when the computation is actually done, we would have a record of each flag and its setting, as the flag was tested. This could then be used to implement the first of the above three ideas.

The second and third possibilities are similar; what we in fact have implemented is version (3), except that since we never change flags in our experiments, we need never clear any tables.

Further complications occur when the tabulated function also produces side effects such as setting global variables or performing input or output. Clearly

"remembering" the result of the last **read** is not an adequate substitute for executing another **read**.

5 Experiments

Several experiments were run using hash tables to tabulate the main functions in the simplifier and differentiator in Macsyma.

Comparing this modified Macsyma to SMP and Maple can be confusing. The algorithms and representations used inside Maple benefit from an early commitment to storing unique versions of expressions. This reduces testing for structural equivalence to a test for pointer equality. The penalty that Maple pays is that creating new expressions is more elaborate than in Macsyma; Maple must see if the expression or part of it already exists. Adding such hash tables to Macsyma would require extensive restructuring. We hope to explain how our evidence supports the conclusion that hashing may not be so good as an add-on feature.

In this section we will discuss the rationale, the benchmarks, and the significance of our measurements.

The performance of Vax Macsyma ("Vaxima") was measured, running under Franz Lisp Opus 42 on a Vax 8600. The Franz Lisp built-in hash tables were used to tabulate two system functions. Structural equivalence was used to test for identical expressions. The tables were used to tabulate the simplifier **simplifya** [5] the symbolic differentiation program **sdiff**. These were chosen because they were intuitively likely candidates, even though they both examine global flags. (The Maple system tabulates the evaluator and Taylor-series expander as well. The Taylor-series expander went virtually unused in our benchmarks, and side-effects in the evaluator caused the tabulated version to return incorrect results. Thus we did not use tabulated versions of **meval** and **$taylor** in our tests.) Another experiment was done with the Macsyma **great** function, which is used to compare, lexically, internal expressions. A major portion of the time in **simplifya** in Macsyma is used to group and rearrange expressions in sums and products. Unfortunately, the **great** function is called so many times on trivial comparisons that tabulating it is not worthwhile. Cutting the computation at the higher level seems more productive.

In the long run we found nominal speedups for three rather specific test cases and slowdowns for all the rest. The sped-up benchmarks were a priori thought to be good candidates; they each involved recursive operations that kept regenerating common subexpressions. Since we could not rule out the possibility that the slowdowns on the other benchmarks were due to a poor implementation of hashing, we extended our measurements to get concrete upper bounds on possible performance improvement.

5.1 Instrumentation

We tried to identify the maximum speed-up available by measuring the total time used by each procedure in the benchmark computations. By subtracting the time spent in **simplifya** and **sdiff** we could see the net reduction in time possible if the time spent in the two procedures could be reduced to zero (by miraculous speedup). These measurements were obtained as follows: Each procedure has a clock. The difference between entry and exit times for top-level calls was added to the clock. A counter was used to determine whether or not invocations of a procedure were top-level or not, by incrementing the counter on call and decrementing it on return. The time spent by non-top-level calls was charged to the procedure implicitly since the clock keeps running as long as the top-level call is active. The presence of the timing macros added a small overhead to the execution. The monitoring overhead amounted to roughly one procedure call/return, one add, one subtract, one compare, and a few stores per invocation of the procedure. The total instrumentation overhead was between 2.6% and 15.6% of the execution time excluding garbage-collection; much of this was not charged to the calls since the majority of the instrumentation computation occurred before and after the top-level calls.

Second, the "redundant time" spent in each procedure was measured. This was the cumulative time spent processing inputs that had already been processed. This was done by running the system twice: once to get timing information, and the second time to use the timing information and the call pattern to estimate the redundancy. This took a large amount of computation. The first run collected the times with as little interference as possible to the program execution; this way the results are more accurate than they would be if the more complex instrumentation were allowed to affect the results.

In the first run, the entry and exit times were recorded for each of the two procedures **simplifya** and **sdiff**. These were concatenated onto a list. This amounts to an overhead of two *cons* operations and two calls to *ptime* per invocation. The list of times consumed some space; garbage-collection times were driven up dramatically. Using this information, we looked at the calls within the system along with their inputs, and figured out which ones were redundant and how much time they took. The instrumentation overhead added between 11.9% and 31.4% to the execution time excluding garbage-collection. Much of this overhead was charged to the calls, but it cancelled out to some degree because the time for redundant calls was being deducted from the total time.

In the second run, we took the list of times from the first run and used them to produce an estimated time. A list of inputs was maintained for each of the two functions, along with a flag indicating whether the function was active or not up the call chain. The time for each top-level *redundant* call was then eliminated from the hypothetical best-case time. The time for a redundant call within a non-redundant call was deducted from the time for the non-redundant one.

The result was the estimated non-redundant time for each function. Subtracting this from the total time for the function gives the estimated redundant time, the time spent reprocessing old inputs. This provided an upper bound on speedups possible from eliminating recomputations on the same inputs.

Thirdly, the "hashed time" was derived. This was done by using the hash tables to save the inputs and outputs, and measuring the amount of time taken by the modified system. This was the prototype hashed Macsyma.

Presumably, for a given procedure we should demonstrate

(total time without hashing) $\geq$ (total time using hashing) - (hashtable overhead) = (essential nonredundant calculation time)

The nonredundant time assumes an ideal lookup scheme with zero overhead. The "total time without hashing" includes the entire computation, including redundant computations. The "Total time using hashing" is what we must analyze. Either

(total time without hashing) $\geq$ (total time using hashing)

or

(total time using hashing) $\geq$ (total time without hashing)

depending upon the level of redundancy and hash table overhead.

Additional notes: The instrumentation is rather crude. The clock used has a resolution of 1/60 seconds, so some fast calls may not appear to take any time at all, although this was statistically unlikely over many calls. The monitoring macros themselves add an overhead to the execution. We tried to minimize this as much as possible, but the balance of time between the different procedures may have shifted slightly because of this.

5.2 Benchmarks

The benchmarks used for testing are called *FG*, *Logs*, *SlowTaylor*, and *Begin*. Listings of these appear in the appendix. Each consists of a sequence of commands. We counted only the cumulative times for all commands. The Macsyma display was turned off in some since printing large expressions dominated some of the computation. In most cases the time for the benchmark is dominated by one large test. We feel this is reasonable because the larger test case is probably the most realistic one. The benchmarks are described as follows:

FG	Generates polynomials in 3 variables (F & G series of celestial mechanics.
Logs	Takes successive derivatives of log(log(log(log(log(x))))).
SlowTaylor	Computes a Taylor-series expansion in an inefficient way.
Begin	Performs a variety of operations.

The *Begin* benchmark (*beginning demonstrations*) which does no obvious redundant calculation shows the least benefit from hashing, and is typical of a number of other benchmarks initially considered.

5.3 The Measurements

Table 1 shows the percentage times spent in each of the critical procedures for each of the benchmarks. There is considerable overlap in the numbers, since if **sdiff** calls **simplifya**, the time will be charged to both. By timing each separately, we obtained an upper bound on the speedup if either were individually sped up so as to take no time at all.

Also in Table 1 is the estimate of nonredundant time as defined in the previous subsection. This projects how much of the time could be eliminated by a perfect (i.e. "free") lookup scheme. This is only a projection, however, since the time overlapped between the two functions can be removed only once. In reality the hashing overhead will take a big slice of this. For example, the FG benchmark spent 90% of its time in **simplifya**. About 19% of its time (90%-70.9%) was spent recomputing known results.

The measured times are indicated in table 2.

Table 1 – Percentage of Time Consumed in Critical Procedures

	Time		Nonredundant Time	
Case	simplifya	sdiff	simplifya	sdiff
FG	90.0	86.1	70.9	65.7
Logs	89.2	100.0	36.6	57.4
SlowTaylor	94.9	100.0	15.6	24.5
Begin	29.3	0.0	26.8	0.0
ignoring garbage-collection time				

Table 2 – Timing Comparison: Unhashed vs. Hashed

	Unhashed		Hashed	
Case	CPU	GC	CPU	GC
FG	15.3	3.0	22.4	5.3
Logs	34.4	9.2	26.0	1.7
SlowTaylor	18.1	4.3	6.9	0.0
Begin	2.6	1.5	3.7	1.5
"cpu" is total time excluding garbage-collection time. all time in seconds on a VAX 8600				

Table Three – Percentages of Possible Speedup		
Case	Potential	Actual
FG	28.9	-46.4
Logs	161.3	32.3
SlowTaylor	481.2	162.3
Begin	5.3	-42.3
ignoring garbage-collection time		

Table 3 presents the final results. The "Potential Speedup" is the percentage speedup if all *redundant* time could have been eliminated. The "Actual Speedup" is the percentage speedup (or slowdown) using the hash tables as we implemented them.

6 Conclusions and Caveats

The results from table 3 are mildly discouraging. At best, a zero-time hashing scheme would produce a speedup factor of 5 for the *SlowTaylor* benchmark, which was designed for the sole purpose of generating an opportunity to remove redundant work. The *Logs* benchmark followed with a factor of 2.6, and the other two benchmarks had little potential speedup. The overhead of maintaining real hash tables reduced these potential performance wins down to a factor of 2.6 for *SlowTaylor* and inconclusive results for everything else.

How likely is it that one will re-simplify an expression? Figures 1 and 2 show the behavior of expressions passing through the simplifier for the *FG* benchmark (log-log scale). Some 5000 expressions are simplified only once. Some 100 expressions are simplified 3 times. Fewer than 10 expressions are simplified 7 times or more. Nevertheless, if these expressions were tough ones, we might win.

Figure 2 attempts to correlate the number of times a given expression is simplified with the complexity of the expression. The number of resimplifications of an expression are plotted against the (average) size of the expression. This was measured by collecting expressions simplified the same number of times and averaging their sizes; this average size decreases dramatically with the number of resimplifications. This suggests that the more likely an expression is to be in the hash table, the less work it would have taken to resimplify it the "hard way". Virtually all expressions simplified more than 10 times were atoms.

These facts suggest that the tables will tend to fill with a large number of small expressions. This will slow down the table management. It is very likely that the table management time is often too expensive for the time saved by tabulating small expressions. The redundancy was much lower for the large expressions, which probably took more computation time. Although the table overhead would be decreased if only large expressions were considered, fewer hits would occur.

Figure 1 -- # of simplifications vs number of expressions

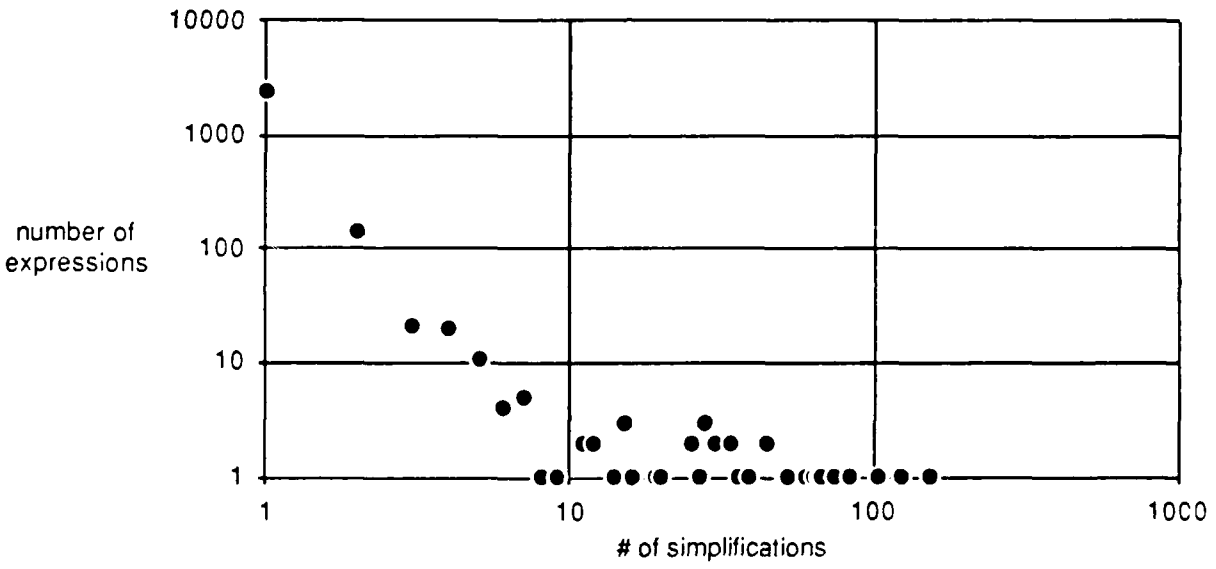


Figure 2 -- # of simplifications vs average size

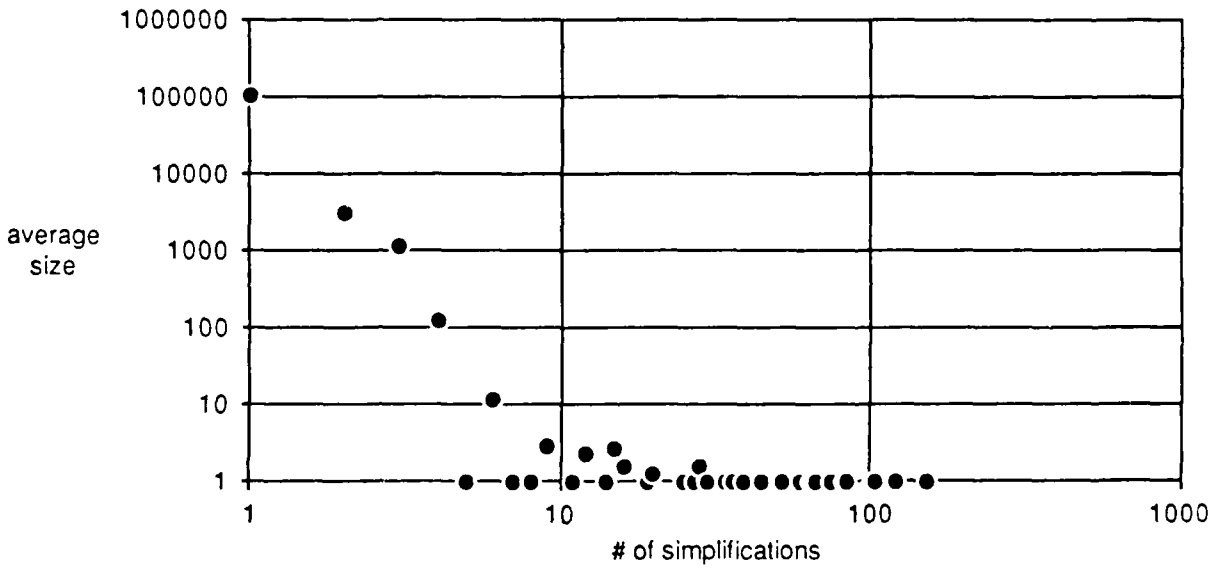
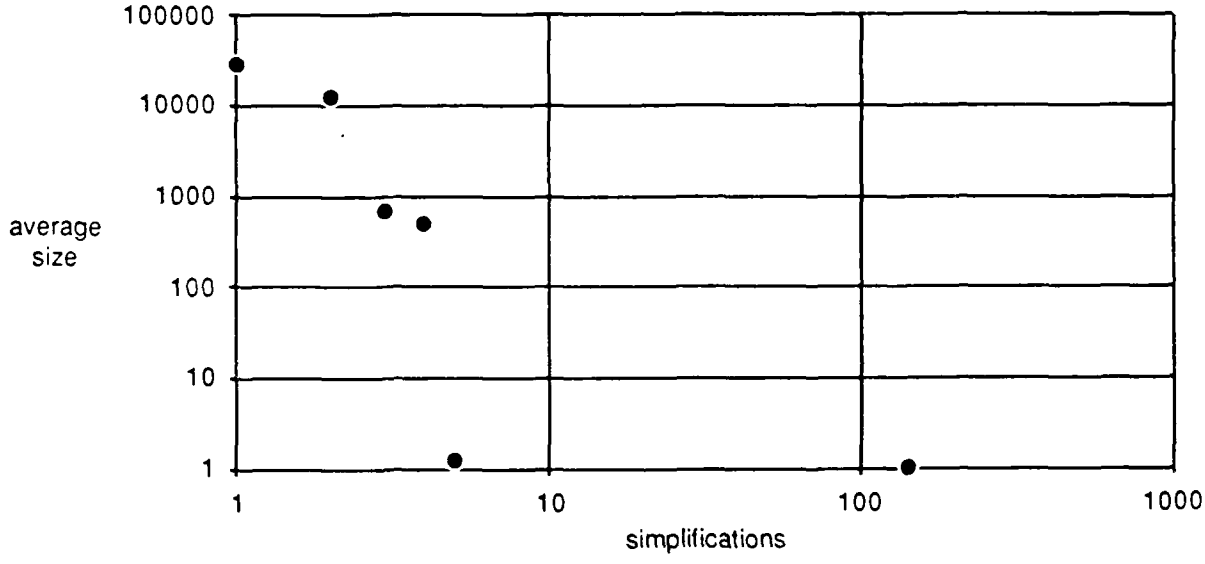


Figure 3 -- # of simplifications vs. average size



The potential speedups from table 3 do suggest that we could try harder. The time spent in the hash tables was very significant. A faster management scheme (especially using a faster hash function) may help. No effort was made to leave "bad" elements out of the table or order the hash bucket contents for faster search. Some change in the internal structure or representations used within Macsyma might improve things as well; the upper bounds we have drawn yield no information about this. It is unfortunate that we have no statistics for a non-hashing Maple to compare against.

The Maple system uses an additional hash table to maintain unique copies of every expression, equivalent to memo-izing the "cons" operation. This is done primarily to conserve space (at the expense of time). A side benefit is that every subexpression is uniquely identified by its address in the table. This address is used to compute the hash codes in the lookup tables, in less time than the Macsyma system can. The cost of generating the hash code is effectively amortized over the construction of new expressions.

We thought it would be worthwhile to test the unique-copies idea in Macsyma. Figure 3 plots the number of expressions simplifying to a given expression against its average size. This was measured by collecting expressions that were the simplified form of n different expressions, and averaging their sizes. It shows that the average size tended to decrease quickly with the number of different expressions simplifying to it, or alternately, that a factor of about 2 in space would be saved if all expressions equivalent under simplification were represented by the same object. It would probably be worth the effort if this were sustained, but this benchmark (FG) would seem to be a likely beneficiary: it wasn't.

The most frequent output of the simplifier was *zero*. The second most frequent was a product of an integer and 2 variables. We suspect that atoms (or small expressions) will generally be, by far, the most likely recurring expressions. The Lisp underlying the Macsyma system already works to maintain unique copies of each atom. We conclude that the following are true:

- Maintaining unique copies of all Maple expressions is probably not much more expensive than maintaining unique copies of atoms, since atoms are the most frequent expressions.
- Using hash tables and structural equivalence to equate objects in Macsyma adds a higher overhead, since effort is already spent to make atoms unique but this property is unused. Building the unique-copy cons mechanism into Lisp (e.g. HLISP [7]) would perhaps put Macsyma on an even footing.
- Hashing is useful only for exceptionally redundant computations. The overhead in Maple is not very significant, so little is lost. The overhead in Lisp systems is higher because the unique-copy computations are being done twice for the most common case (i.e. atoms).

7 An Application of Parallelism

On a parallel processor, searching the lookup table could commence in parallel with the computation. This would eliminate the extra overhead from the computation and would provide speedup whenever a successful table search is faster than recomputing the function. Similarly, putting entries into the table and rehashing can be performed in parallel with other parts of the computation. Except for possible memory contention, this is almost assured of breaking even or winning. This idea should be pursued.

Another possibility to explore is precomputing elements and putting them into the table *before* they are requested. The risk here is that the precomputed result may never get used. One idea would be to automatically expand, factor, or simplify an expression when it is produced. Another would be to operate within different contexts, such as evaluating an integral under different assumptions. Partial results might be saved which could be requested later, such as the indefinite integral of an expression when the definite one was requested or the derivative of an expression when the limit is being taken. Different simplified forms could be produced, such as producing one where symbolic constants (such as e) are replaced with numeric approximations, or left in symbolic form. These jobs are somewhat dubious, and probably fall in the category of making work for otherwise idle processors. Finally, we would like to emphasize that in systems with a highly complex global state this technique may be difficult to apply. Only "pure" side-effect-free functions are good candidates.

8 Acknowledgments

This work was supported in part by the Army Research Office, grant DAAG29-85-K-0070, through the Center for Pure and Applied Mathematics, University of California, Berkeley, and the Defense Advanced Research Projects Agency (DoD) ARPA order #4871, monitored by Space & Naval Warfare Systems Command under contract N00039-84-C-0089, through the Computer Science Division, University of California, Berkeley.

References

[1] Abelson, H., Sussman, G.J., Sussman, J. (1985). *Structure and Interpretation of Computer Programs*, MIT Press/ McGraw-Hill Book Co., New York, N.Y.

[2] Bentley, J.L. (1982). *Writing Efficient Code*, Prentice-Hall, Englewood Cliffs, N.J.

[3] Buchberger, B., Loos, R. (1983). Algebraic Simplification. In B. Buchberger et al., *Computer Algebra (second edition)*, Springer-Verlag. 11-44

[4] Char. B.W. et al. (June 1984). *On the Design and Performance of the Maple System.* Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, CS-84-13.

[5] Fateman, R.J. (June 1979). Macsyma's General Simplifier: Philosophy and Operation. In Lewis, V.E. (ed), *Proceedings of the 1979 Macsyma Users Conference.* Washington D.C. 563-582.

[6] Greif, J. (1985). The SMP Pattern Matcher. In B.F. Caviness (ed), *Proc. Eurocal '85*, vol. 2, Lecture Notes in Computer Science 204, Springer-Verlag, 303-314

[7] Goto, E., Kanada, Y. (1976). Hashing Lemmas on Time Complexities with Applications to Formula Manipulation. *Proc. SYMSAC '76*, ACM, New York, 1976, 154-158.

[8] Michael B. Monagan, private communication 6/16/87

[9] Ponder, C., Fateman, R. (1987). A Short Note on Program Profiling. Submitted to *Software - Practice and Experience*

9 Appendix: The Test Cases

9.1 The *FG* Benchmark

```
showtime:all$

/* f&g general representation */

gradef(mu,t,-3*mu*sigma)$
gradef(sigma,t,eps-2*sigma**2)$
gradef(eps,t,-sigma*(mu+2*eps))$
f[0]:1$
g[0]:0$
f[i]:= -mu*g[i-1]+diff(f[i-1],t)$
g[i]:= f[i-1]+diff(g[i-1],t)$
expop:1$

f[10]$
g[10]$
f[15]$
g[15]$

expop:0$

kill(f,g);
```

9.2 the *logs* benchmark

```
showtime:all$

f[0]:x$
f[n]:=log(f[n-1])$
diff(f[5],x)$
diff(%,x)$
diff(%,x)$
diff(%,x)$
diff(%,x)$
diff(%,x)$

kill(f);
```

9.3 the *slowtaylor* benchmark

```
slowtaylor(expr,var,point,hipower):=
     block([result],
   result:at(expr,var=point),
   for i:1 thru hipower
       do (result:result+(var-point)^i* at(diff(expr,var,i)/i!, var=point)),
   result)$

showtime:all$


slowtaylor(tan(sin(x))-sin(tan(x)),x,0, 7);
```

9.4 the *begin* benchmark

```
showtime:all;
1/(x^3+2);
diff(%,x);
ratsimp(%);
taylor(sqrt(1+x),x,0,5);
%*%;
(x+3)^20;
rat(%);
diff(%,x);
factor(%);
factor(x^3+x^2*y^2-x*z^2-y^2*z^2);
solve(x^6-1);
mat:matrix([a,b,c],[d,e,f],[g,h,i]);
%^^2;
fac(n):=if n=0 then 1 else n*fac(n-1);
fac(5);
g(n):=sum(i*x^i,i,0,n);
g(10);
```

END

DATE

FILMED

DTIC

9-88